PR #34081 完整报告

sgl-project/sglang

config: business code no longer reads the published ServerArgs

合并时间：2026-08-10 05:44

原文链接：http://prhub.com.cn/sgl-project/sglang/pull/34081

执行摘要

- 一句话：配置读取全迁 runtime_context，ratchet 基线归零
- 推荐动作：值得精读。重点学习：runtime_context 访问器层的设计（bag 派生与 slot 透传的取舍）、ratchet 的 AST 扫描与词法作用域处理（模块别名、嵌套函数、动态 getattr）、遮蔽守卫对全部绑定形式的枚举，以及缓存生命周期在 publish/reset 中的管理。这些模式对大型 Python 代码库的配置演进有直接借鉴价值。

功能与动机

PR body 明确目标：『The last field reads outside the resolution pipeline are gone』，即业务代码不再读取已发布的 ServerArgs。此前残留的两类读取（派生成员、被 live 拓扑遮蔽的配置尺寸）都需要『命名归宿』，而 runtime_context 是 slot 的持有者，因此成为这些访问器的唯一家园。ratchet 的豁免从名字列表改为模块所有者，防止新读取隐藏在熟悉的成员名后面。

实现拆解

1. 新增命名访问器层：python/sglang/srt/runtime_context.py 增加 6 个派生成员访问器与 4 个 configured_*_size() 访问器。凡输入全部是已发布 bag 叶子的（spec、schedule、exec.kernel）直接从 bag 派生，跟随 post-publish override；真正依赖 ServerArgs 计算或进程 model config 的（mamba_cache_chunk_size、uses_mla_backend、process_model_config）保持只读透传，由 runtime_context 独占读取 slot。configured_*_size 通过 _configured_parallel 读取 parallel bag，绕过 ParallelContext 的 live 属性遮蔽。
2. 迁移业务调用点：18 处调用点改为命名访问器。典型案例：sarvam_moe.py 的 _set_current_attention_backend 改用 attention_backends() 且 radix cache 判断改读 get_memory()；schedule_batch.py 与 hybrid_linear_attn_backend.py 的 mamba 状态缓存点改 mamba_cache_chunk_size()；kimi_k3.py/kimi_k25.py 的 IPC refcount 改 configured_tp_size()（与 tokenizer 进程的回收器对齐）；flashinfer_cutedsl.py 改 cutedsl_moe_max_num_tokens()；cp/base.py 的 DSA 谓词改 get_parallel().enable_prefill_cp。
3. 收紧配置读取守卫：test_global_config_read_ratchet.py 将豁免改为模块所有者集合 _SLOT_OWNERS（runtime_context、server_args、arg_groups），direct/alias 基线设为 0；扫描扩展识别 getattr 动态读取、模块级别名（含实例属性形式）、别名拷贝到

fixpoint，并修正词法作用域遮蔽——每个读按 enclosing function chain 判定，嵌套函数绑定不再污染外层。configured_*_size 的 7 个调用点与原因登记进 _CONFIGURED_SIZE_CALL_SITES，被测试断言精确匹配。

4. 新增遮蔽守卫：test_context_accessor_shadowing.py 用 AST 扫描整个包，检查函数内局部绑定是否遮蔽已导入的访问器（会触发 UnboundLocalError）；覆盖全部绑定形式：赋值、for、with、walrus、except、解构、嵌套 def/class 名，并按词法作用域跟踪函数级 import 的可见性。
5. 配套测试与清理：test_runtime_context.py 新增跨层一致性测试（bag 访问器与 ServerArgs 成员在所有输入组合上相等）、自适应 bound 生命周期测试（republish/reset 后缓存失效重算）、访问器包装语义静态测试（方法必须调用、属性不能调用）。同时删除 9 处死绑定与失效测试 mock，server_args.py 的 get_attention_backends、enable_mamba_extra_buffer 等委托 arg_groups.overrides 的 *_of helper，消除重复实现。

关键文件：

- python/sglang/srt/runtime_context.py（模块 上下文；类别 source；类型 dependency-wiring；符号 mamba_cache_chunk_size, max_speculative_num_draft_tokens, adaptive_draft_token_bound, uses_mla_backend）：核心变更文件：新增派生成员访问器与 configured*_size 访问器，并在 set_server_args/reset_context 中管理自适应 draft bound 缓存生命周期，是唯一允许读取 ServerArgs slot 的模块。
- test/registered/unit/test_global_config_read_ratchet.py（模块 配置守卫；类别 test；类型 test-coverage；符号 collect, _getattr_name, _is_global_call, TestConfiguredSizeCallSites）：读取守卫的核心测试：豁免从按名字改为按模块，direct/alias 基线归零，并扩展识别 getattr、模块级别名、词法作用域遮蔽，登记 configured*_size 调用点原因。
- test/registered/unit/test_context_accessor_shadowing.py（模块 遮蔽守卫；类别 test；类型 test-coverage；符号 _module_level_accessor_imports, _bound_names, _own_scope_statements, _child_functions）：新增 AST 守卫，防止机械迁移产生的函数级局部变量遮蔽访问器导致 UnboundLocalError 回潮，是本次重构的关键防回归设施。
- test/registered/unit/test_runtime_context.py（模块 上下文测试；类别 test；类型 test-coverage；符号 TestDerivedPredicatesAgreeAcrossTiers, TestAdaptiveDraftBoundLifecycle, TestNamedAccessorsCallWhatTheyWrap, _write_config）：新增三组测试：跨层一致性（bags 与 ServerArgs 成员）、自适应 bound 生命周期、访问器包装语义，直接钉死新访问器契约。
- python/sglang/srt/server_args.py（模块 启动参数；类别 source；类型 dependency-wiring；符号 get_attention_backends, enable_mamba_extra_buffer, enable_mamba_extra_buffer_lazy）：解析管线自身的成员方法改为委托 arg_groups.overrides 的 *_of helper（attention_backends_of、mamba_extra_buffer_of），消除与 bag 派生的重复实现。
- python/sglang/srt/models/sarvam_moe.py（模块 模型层；类别 source；类型 dependency-wiring；符号 _set_current_attention_backend, _run_mha_prefill）：业务调

用点迁移的典型样板：_set_current_attention_backend 与 radix cache 判断从 ServerArgs 实例字段改为命名访问器，展示新读取方式。

- python/sglang/srt/managers/schedule_batch.py（模块 调度器；类别 source；类型 dependency-wiring；符号 _mamba_radix_cache_v2_req_prepare_for_extend）：mamba radix cache v2 路径从 get_server_args() 读取 chunk size 改为 mamba_cache_chunk_size() 访问器，是访问器迁移在调度批处理中的代表。

关键符号：mamba_cache_chunk_size, max_speculative_num_draft_tokens, _adaptive_draft_token_bound, uses_mla_backend, attention_backends, process_model_config, cutedsl_moe_max_num_tokens, _configured_parallel, configured_tp_size, configured_pp_size, configured_moe_dp_size, configured_attn_cp_size, RuntimeContext.set_server_args, RuntimeContext.reset_context, _shadowing_assignments, _collect, TestAdaptiveDraftBoundLifecycle.test_republishing_recomputes_the_bound, SarvamMoE._set_current_attention_backend, ScheduleBatch._mamba_radix_cache_v2_req_prepare_for_extend

关键源码片段

python/sglang/srt/runtime_context.py

核心变更文件：新增派生成员访问器与 configured_*_size 访问器，并在 set_server_args/reset_context 中管理自适应 draft bound 缓存生命周期，是唯一允许读取 ServerArgs slot 的模块。

```
# --- Derived config accessors ------------------------------------------------
# 以下值由多个配置字段加 HF config 推导，无法成为 namespace 叶子；
# runtime_context 是 slot 的持有者，因此这里是它们唯一的『命名归宿』。
# 每个访问器保持 ServerArgs 成员的精确语义（包括总是基于进程即目标
# 模型的 model config 推导）。

def mamba_cache_chunk_size() -> int:
    """mamba 状态缓存点粒度：max(模型 mamba chunk size, page_size)。"""
    # 该成员是属性且只在发布后读取，直接透传 slot。
    return get_server_args().mamba_cache_chunk_size


def max_speculative_num_draft_tokens() -> int | None:
    """投机解码可用的最大 draft-token 数。

    三个输入都是 spec bag 的叶子，所以从袋读取并跟随 post-publish
    override；ServerArgs 上的同名成员是解析管线用的发布前等价物。
    """
    spec = get_spec()
    if spec.speculative_num_draft_tokens is None:
        return None
    if not spec.speculative_adaptive:
        return spec.speculative_num_draft_tokens
```

```
    # 自适应分支要解析 JSON 配置，且此函数按 decode batch 调用
    # （spec_prepare_for_decode），所以按输入（cfg_path）做 lru_cache，
    # 而不是缓存一次，这样 post-publish override 仍会重算。
    return _adaptive_draft_token_bound(spec.speculative_adaptive_config)


@functools.lru_cache(maxsize=8)
def _adaptive_draft_token_bound(cfg_path: str | None) -> int:
    # 自适应 spec 目前要求 topk=1，所以每个运行时状态需要 steps + 1
    # 个 draft-token 槽位，与 ServerArgs 成员镜像一致。
    from sglang.srt.speculative.adaptive_spec_params import (
        resolve_candidate_steps_from_config,
    )
    candidate_steps = resolve_candidate_steps_from_config(cfg_path=cfg_path)
    return max(candidate_steps) + 1


def uses_mla_backend() -> bool:
    """本进程模型是否走 MLA 注意力路径。"""
    return get_server_args().use_mla_backend()


def attention_backends() -> tuple:
    """配置的 (prefill, decode) 后端对，split 字段回退到 attention_backend。

    三个输入都是 exec.kernel 叶子，所以从袋读取并跟随 post-publish
    override；ServerArgs.get_attention_backends 是解析管线的发布前等价物。
    """
    from sglang.srt.arg_groups.overrides import attention_backends_of
    # 三个叶子在同一 bag 中，直接复用解析管线的 helper —— 回退规则只有一份。
    return attention_backends_of(get_exec().kernel)


def _configured_parallel(name: str):
    """读 parallel bag 本身而非 ParallelContext —— live property 会遮蔽
    tp/pp/moe_dp/attn_cp 四个名字。parallel 不在 per-role 命名空间表内，
    所以不能走 config_bag() 的角色检查。
    """
    config = _CONTEXT.parallel._config
    if config is None:
        raise ValueError('config namespace parallel not published')
    return getattr(config, name)


def configured_tp_size() -> int:
    return _configured_parallel('tp_size')
```

test/registered/unit/test_context_accessor_shadowing.py

新增 AST 守卫，防止机械迁移产生的函数级局部变量遮蔽访问器导致 UnboundLocalError 回潮，是本次重构的关键防回归设施。

```
  def _bound_names(target):
      """一个绑定目标引入的所有名字，解构展开。

      a, (b, c) = ... 与 for x, y in ... 会穿过 Tuple/List/Starred 节点，
      所以只接受裸 ast.Name 的检查会漏掉它们。
      """
      if isinstance(target, ast.Name):
          yield target.id
      elif isinstance(target, ast.Starred):
          yield from _bound_names(target.value)
      elif isinstance(target, (ast.Tuple, ast.List)):
          for element in target.elts:
              yield from _bound_names(element)


  def _own_scope_statements(node) -> tuple:
      """该函数自身作用域的语句，以及每个嵌套 def/class 的 (name, lineno)。

      嵌套定义的 *名字* 是当前作用域的绑定（先调用后定义会抛
      UnboundLocalError），但其 *函数体* 属于嵌套作用域，下降访问会错误归属。
      """
      own_scope = []
      nested_def_bindings = []
      pending = list(node.body)
      while pending:
          stmt = pending.pop()
          if isinstance(stmt, (ast.FunctionDef, ast.AsyncFunctionDef, ast.ClassDef)):
              nested_def_bindings.append((stmt.name, stmt.lineno))
              continue
          if isinstance(stmt, ast.Lambda):
              continue
          own_scope.append(stmt)
          pending.extend(ast.iter_child_nodes(stmt))
      return own_scope, nested_def_bindings


  def _shadowing_assignments(tree: ast.AST, module_accessors: set[str]):
      """报告『函数内局部绑定遮蔽了该作用域可见的访问器』。

      Python 依据函数内 *任意* 绑定判定名字为局部，因此 for 变量、
      with ... as、walrus、推导式目标、except ... as 都与赋值一样
      会遮蔽访问器。可见性遵循词法作用域：模块级 import 可达所有函数，
      函数内 import 只达自身与嵌套函数，不会误报无关兄弟函数。
      """
      stack = [(fn, module_accessors) for fn in _child_functions(tree.body)]
      while stack:
```

```
    node, inherited = stack.pop()
    own_scope, nested_def_bindings = _own_scope_statements(node)
    local_imports = {
        alias.asname or alias.name
        for stmt in own_scope
        if isinstance(stmt, ast.ImportFrom) and stmt.module == _CONTEXT_MODULE
        for alias in stmt.names
    }
    visible = inherited | local_imports
    # 内嵌 def/class 的名字绑定在本作用域，等同赋值。
    for name, lineno in nested_def_bindings:
        if name in visible:
            yield node.name, name, lineno
    for inner in own_scope:
        targets = []
        # 收集所有会绑定局部的语句形式。
        if isinstance(inner, ast.Assign):
            targets = inner.targets
        elif isinstance(inner, (ast.AnnAssign, ast.AugAssign)):
            targets = [inner.target]
        elif isinstance(inner, (ast.For, ast.AsyncFor, ast.comprehension)):
            targets = [inner.target]
        elif isinstance(inner, ast.NamedExpr):
            targets = [inner.target]
        elif isinstance(inner, (ast.With, ast.AsyncWith)):
            targets = [i.optional_vars for i in inner.items if i.optional_vars]
        elif isinstance(inner, ast.ExceptHandler) and inner.name:
            targets = [ast.Name(id=inner.name, ctx=ast.Store())]
        for target in targets:
            # 解构绑定需要递归展开 Tuple/List/Starred。
            for name in _bound_names(target):
                if name in visible:
                    yield node.name, name, getattr(inner, 'lineno', node.lineno)
    # 把『本作用域可见集合』传给子函数，继续深度优先扫描。
    for nested in _child_functions(node.body):
        stack.append((nested, visible))
```

评论区精华

Codex 提出两处 P1 并获修复：cutedsl_moe_max_num_tokens 访问器返回 bound method 会引发 TypeError（作者确认后新增 TestNamedAccessorsCallWhatTheyWrap 静态检查推广到所有访问器）；自适应 draft-token bound 在 *decode* 热路径反复读取 JSON（作者改为 lru_cache 按 cfg_path 键控，2000 次调用 0.7 ms）。多轮 P2 聚焦两个 AST 守卫的作用域语义：ratchet 的模块别名从文件级 union 改为词法作用域判定，遮蔽守卫补全 for/with/walrus/except/ 解构等所有绑定形式，且嵌套 def 名称计入外层绑定。遗留未解决疑虑：sarvam_moe 改为进程级全局访问器后多 Engine 场景可能读到其他 Engine 的配置；preserve_config/override restore 路径未恢复 adaptive bound 缓存。

- cutedsl_moe_max_num_tokens 访问器返回 bound method 导致 TypeError (correctness): 作者确认并让访问器调用成员；同时新增 TestNamedAccessorsCallWhatTheyWrap 静态检查所有 return get_server_args().X 形态，断言方法必须调用、属性 / cached_property 不能调用。
- 自适应 draft-token bound 在 decode 热路径反复读 JSON (performance): 作者用 functools.lru_cache(maxsize=8) 按 cfg_path 键控 memoize；测量 2000 次调用 0.7 ms。
- 自适应 bound 缓存跨 publish/reset 生命周期残留 (correctness): 在 RuntimeContext.set_server_args 与 reset_context 中 _adaptive_draft_token_bound.cache_clear()；由 TestAdaptiveDraftBoundLifecycle 覆盖 republish 与 reset 两个场景。
- ratchet 模块级别名按文件级 union 会漏检全局读取 (correctness): 改为按词法作用域判定：parent map 给每个读定位 enclosing function chain，只有作用域链上的绑定才算遮蔽；反向验证模块级读、全局读、局部遮蔽三种场景。
- ratchet 嵌套函数绑定被 ast.walk 错误归属到外层 (correctness): 收集绑定时不下降嵌套 FunctionDef/AsyncFunctionDef/Lambda/ClassDef 体；每个作用域只归属自己的 store。
- accessor shadowing 守卫漏掉 for/with/walrus/except/ 解构等绑定形式 (correctness): 扩展收集全部绑定语句形式，并用递归 target 展开处理 Tuple/List/Starred 解构；函数内 re-import 同一 callable 显式豁免并写进 docstring。
- 嵌套函数名也算外层绑定（def get_exec 遮蔽） (correctness): _own_scope_statements 把嵌套 def/class 名字计入外层绑定，同时不下降其函数体。
- sarvam_moe 多 Engine 场景配置隔离疑虑 (design): 无后续回复，属未解决疑虑；建议按 Engine 归属实例配置或由 runner 传递已解析后端。
- preserve_config/override restore 生命周期未恢复 adaptive bound 缓存 (correctness): 无作者回复，未解决；建议 restore 时也恢复 memo 或按 publication 记录缓存。
- kimi IPC refcount 与 recycler 的 configured_tp_size 不一致 (correctness): kimi_k25 与 kimi_k3 均改为 configured_tp_size()，两个调用点登记进 _CONFIGURED_SIZE_CALL_SITES 并注明原因；测试改为 publish 配置而非 fake getter。

风险与影响

- 风险：多 Engine 配置隔离风险：sarvam_moe.py 的注意力后端与 radix cache 判断改读进程级最新发布配置，若多个 in-process Engine 使用不同 attention backend，可能出现后端选择与 get_attn_forward_method 不一致的执行路径（Codex P1，未关闭）。缓存生命周期风险：_adaptive_draft_token_bound 是进程级 lru_cache，preserve_config() 与 _ServerArgsOverride.restore() 绕过 set_server_args 直接恢复 bags 时缓存不会重建，文件被改写或删除时可能失败或用旧值（无回复）。AST 扫描局限：ratchet 看不到运行时名字 getattr(sa, name)、容器 / 跨对象间接引用、跨作用域拷贝；遮蔽守卫对函数内重新 import 同一 callable 显式豁免，若未来语义分化可能误报。configured vs live 语义：configured_*_size 读 bag 而非 record，bag 未发布时抛 ValueError，调用时机（Indexer 构造、tokenizer 进程）必须保持正确顺序。回归面：35 文件、18 调用点迁移触及 mamba radix cache、speculative decoding、MLA backend、Kimi 多模态 IPC、CuteDSL MoE A2A 等核心路径，单测共 583 项通过，但部分路径依赖 e2e 或动态触发。

- 影响：对用户与运维：ServerArgs 解析后成为只读记录，运行时改配置必须走 override 与配置袋，行为契约变化。对开发者：新增配置读取必须通过 runtime_context 访问器或 bags，ratchet 强制执行；configured_*_size 语义需要登记原因。对系统：消除业务代码对发布记录的字段读取，配置读取统一到单一真源，为多实例隔离与可复现解析奠基。对团队：需要理解模块所有者豁免和词法作用域扫描规则，相关测试成为 CI 守门员。
- 风险标记：核心配置路径重构，跨模块 35 文件，AST 静态扫描防回归，缓存生命周期复杂，多引擎配置隔离疑虑

关联脉络

- PR #34096 config: the KV-cache configurator reads the bags: 同一配置袋迁移系列的前置 PR，KV 缓存配置器已改读 bags，本 PR 是该系列中把剩余字段读取清零的收尾。
- PR #34095 config: the runner and scheduler read resolved config from the bags: 同一系列，runner/scheduler 的配置读取已迁到 bags，与本 PR 的访问器迁移直接衔接。
- PR #34094 config: pin that resolution is reproducible from the raw input: 钉死 ServerArgs 解析可重现性契约，与本 PR 的 ratchet 基线归零互为表里。
- PR #34133 config: derive the runner's DCP topology from its ParallelState: 并行拓扑改为由 ParallelState 派生，与本 PR 的 configured_*_size 一起清理并行拓扑的配置读取来源。
- PR #34097 docs(skill): record where config is read now that the seed is off limits: 文档记录配置读取新规矩，帮助后续开发者遵循同一架构。